Foundational proof certificates in first-order logic

Zakaria Chihani, Dale Miller, Fabien Renaud

Foundational proof certificates in first-order logic

Zakaria Chihani, Dale Miller, and Fabien Renaud

INRIA and LIX, Ecole Polytechnique, Palaiseau, France

Abstract. It is the exception that provers share and trust each others proofs. One reason for this is that different provers structure their proof evidence in remarkably different ways, including, for example, proof scripts, resolution refutations, tableaux, Herbrand expansions, natural deductions, etc. In this paper, we propose an approach to *foundational proof certificates* as a means of flexibly presenting proof evidence so that a relatively simple and universal proof checker can check that a certificate does, indeed, elaborate to a formal proof. While we shall limit ourselves to first-order logic in this paper, we shall not limit ourselves in many other ways. Our framework for defining and checking proof certificates will work with classical and intuitionistic logics and with proof structures as diverse as resolution refutations, matings, and natural deduction.

1 Introduction

Consider a world where the multitude of computational logic systems—theorem provers, model checkers, type checkers, static analyzers, etc.—can trust each other's proofs. Such a world can be constructed if computational logic systems can output their proof evidence in documents with a clear semantics that can be validated by a trusted checker. By the term *proof certificate* we shall mean documents that contain the evidence of proof generated by a theorem prover. In this paper, we propose a framework for defining the semantics of a wide range of proof evidence using proof-theoretic concepts. As a result, we refer to this approach to defining certificates as "foundational" since it is based not on the technology used to construct a specific theorem prover but rather on basic insights into the nature of proofs provided by the modern literature of proof theory.

The key concept that we take from proof theory is that of *focused proof systems* [1, 15, 16]. Such proof systems exist for classical, intuitionistic, and linear logics and they are composed of alternating *asynchronous* and *synchronous* phases. These two phases allow for a natural interaction to be set up between a process that is attempting to build a proof (the checker) and the information contained in a certificate. During the asynchronous phase of proof construction, the checker proceeds without reference to the actual certificate since this phase consists of invertible inference rules. During the synchronous phase, information from the certificate can be extracted to guide the construction of the focused proof. The definition of how to check a proof certificate essentially boils down to defining the details of this interaction.

The main structure for our framework contains the following components.

– The *kernel* of our checker is a logic program specification of the *focusing framework LKU* proof system [16]. Since this implementation of *LKU* is high-level and direct, we can have a high degree of confidence that the program does, in fact, capture the *LKU* proof system.
– By restricting various structural rules, *LKU* can be made into a focused proof system for classical logic, for intuitionistic logic, and for multiplicative-additive linear logic. The specifications of these restrictions are contained in separate small *logic definition* documents.
– The kernel implementation of *LKU* actually adds another premise to every inference rule: in particular, the asynchronous rules get a premise involving a *clerk predicate* that simply manages some bookkeeping computations while the synchronous rules get a premise involving an *expert predicate* that extracts information from the certificate to provide to the inference rule. A *proof certificate definition* is a document that defines these two kinds of predicates as well as a translation function from theorems of the considered system to equiprovable LKU formulas.
– A *proof certificate* is a document consisting of the structured object containing the proof evidence supporting theoremhood for a particular formula.

To illustrate this architecture, we present a number of different proof certificates. For example, a certificate for resolution refutations can be taken as a list of clauses (including those arising from the original theorem and those added during resolutions) and a list of triples that describes which two clauses resolve to yield a third clause. Such an object should be easy to produce for any theorem prover that uses binary resolution (with implicit factoring). By then adding to the kernel the logic definition for classical logic (given in [16]) and the definitions of the clerk and expert predicates (given in Section 4.3), resolution refutations can be checked. The exact same kernel (this time restricted to intuitionistic logic) can be used to check natural deduction proofs (*i.e.*, simply and dependently typed λ-terms): all that needs to be changed is the definition of the clerk and expert predicate definitions.

Before presenting specific examples of proof certificate definitions for first-order classical logic in Section 4, we describe focused proof systems in the next section and, in Section 3, we describe how we have augmented and implemented that proof system within logic programming. The current implementation of our proof checking system is available at https://team.inria.fr/parsifal/proofcert/.

2 Proof theory architecture

The sequent calculus of Gentzen [10] (which we assume is familiar to the reader) is an appealing setting for starting a discussion of proof certificates. First of all, sequent calculus is well studied and applicable to a wide range of logics. The introduction rules, structural rules (weakening and contraction), and the identity rules (initial and cut) provide a convincing collection of "atoms" of inference. Additionally, cut-elimination theorems are deep results about sequent calculus proof systems that not only prove them to be consistent but also offers *cut-free*

proofs as a normal form for proof. Girard's invention of linear logic [11] provides additional extensions to our understanding of the sequent calculus, including such notions as *additive* and *multiplicative* connectives, *exponentials*, and *polarities*. Finally, this foundation of Gentzen and Girard lifts naturally and modularly to higher-order logic and to inductive and coinductive fixed points (such as Baelde's μMALL [4]). In this paper, we shall concentrate on first-order (and propositional) logic: we leave the development of proof certificates for higher-order quantification and fixed points for later work.

The sequent calculus has a serious downside, however: sequent proofs are far too unstructured to directly support almost any application to computer science. What one needs is a flexible way to organize the "atoms of inference" into much larger and rigid "molecules of inference." The hope would be, of course, that these larger inference rules can be structured to mimic the notion of proof found in computational logic systems. For example, early work on the proof-theoretic foundations of logic programming [19] showed how sequent calculus proofs representing logic programming executions could be built using two alternating phases: the *backchaining* phase is a focused application of left-rules and the *goal-reduction* phase is a collection of right-rules. Andreoli [1] generalized that earlier work by introducing *focused proofs* for linear logic in which such phases were directly captured and generalized. Subsequently, Liang & Miller presented the *LKF* and *LJF* focused proof systems for classical and intuitionistic logics [15] and later the *LKU* proof system [16] that unified *LKF* and *LJF*. While our current approach to foundational proof certificates is based on *LKU* (allowing the checking of proofs in classical as well as intuitionistic logic), we shall illustrate our approach by considering the simpler *LKF* subsystem of *LKU*. Before presenting *LKF* in detail, we consider the following few elements of its design.

Additive vs multiplicative rules. We shall use t, f, $\wedge$, $\vee$, $\forall$, and $\exists$ as the logical connectives of first-order classical logic and sequents will be one-sided. As is familiar to those working with the sequent calculus, there is a choice to make between using the additive and multiplicative versions of the binary connective $\wedge$ and $\vee$ (and their units t and f, respectively): the most striking difference between these two versions is illustrated with $\vee$:

$$\text{Additive: } \frac{\vdash \Theta, B_i}{\vdash \Theta, B_1 \vee B_2}\ i \in \{1,2\} \qquad \text{Multiplicative: } \frac{\vdash \Theta, B_1, B_2}{\vdash \Theta, B_1 \vee B_2}$$

These two inference rules are inter-admissible in the presence of contraction and weakening. For this reason, one usually selects one of these inference rules and discards the other one. In isolation, however, these inference rules are strikingly different: the multiplicative version is invertible while the additive version reveals that one disjunct is not needed at this point of the proof. The *LKF* proof system will contain the additive and multiplicative versions of disjunction, conjunction, truth, and false: their presence will improve our flexibility for describing proofs.

Polarized connectives. We *polarize* the propositional connectives as follows: those inference rules that are invertible introduce the negative version of the connective while those inference rules that are not necessarily invertible introduce the

$$\frac{}{\vdash \Theta \Uparrow t^-, \Gamma} \quad \frac{\vdash \Theta \Uparrow A, \Gamma \;\; \vdash \Theta \Uparrow B, \Gamma}{\vdash \Theta \Uparrow A \wedge^- B, \Gamma} \quad \frac{\vdash \Theta \Uparrow \Gamma}{\vdash \Theta \Uparrow f^-, \Gamma} \quad \frac{\vdash \Theta \Uparrow A, B, \Gamma}{\vdash \Theta \Uparrow A \vee^- B, \Gamma} \quad \frac{\vdash \Theta \Downarrow [t/x]B}{\vdash \Theta \Downarrow \exists x.B}$$

$$\frac{}{\vdash \Theta \Downarrow t^+} \quad \frac{\vdash \Theta \Downarrow B_1 \;\; \vdash \Theta \Downarrow B_2}{\vdash \Theta \Downarrow B_1 \wedge^+ B_2} \quad \frac{\vdash \Theta \Downarrow B_i \;\; i \in \{1,2\}}{\vdash \Theta \Downarrow B_1 \vee^+ B_2}$$

$$\frac{\vdash \Theta \Uparrow [y/x]B, \Gamma \;\; y \text{ not free in } \Theta, \Gamma, B}{\vdash \Theta \Uparrow \forall x.B, \Gamma} \quad \frac{}{\vdash \neg P_a, \Theta \Downarrow P_a}\ \mathit{init} \quad \frac{\vdash \Theta \Uparrow B \;\; \vdash \Theta \Uparrow \neg B}{\vdash \Theta \Uparrow \cdot}\ \mathit{cut}$$

$$\frac{\vdash \Theta, C \Uparrow \Gamma}{\vdash \Theta \Uparrow C, \Gamma}\ \mathit{store} \quad \frac{\vdash \Theta \Uparrow N}{\vdash \Theta \Downarrow N}\ \mathit{release} \quad \frac{\vdash P, \Theta \Downarrow P}{\vdash P, \Theta \Uparrow \cdot}\ \mathit{decide}$$

Here, P is a positive formula; N a negative formula; P_a a positive literal; C a positive formula or negative literal; and $\neg B$ is the negation normal form of the negation of B.

Fig. 1. *LKF*: a focused proof systems for classical logic

positive version of the connective. Thus the additive rule above for the disjunction introduces $\vee^+$ while the multiplicative rule introduces $\vee^-$. The universal quantifier is obviously polarized negatively while the existential quantifier is polarized positively. Literals must also be polarized: these can be polarized in an arbitrary fashion as long as complementing a literal also flips its polarity. We say that a non-literal formula is positive or negative depending only on the polarity of its top-level connective.

Phases organize groups of inference rules. The inference rules for *LKF* are given in Figure 1. Notice that these inference rules involve sequents of the form $\vdash \Theta \Uparrow \Gamma$ and $\vdash \Theta \Downarrow B$ where Θ is a multiset of formulas, Γ is a list of formulas, and B is a formula. Such sequents can be approximated as the one-sided sequents $\vdash \Theta, \Gamma$ and $\vdash \Theta, B$, respectively. Furthermore, introduction rules are applied to either the first element of the list Γ in the $\Uparrow$ sequent or the formula B in the $\Downarrow$ sequent. This occurrence of the formula B is called the *focus* of that sequent. Proofs in *LKF* are built using two kinds of alternating *phases*. The *asynchronous* phase is composed of invertible inference rules and only involves $\Uparrow$-sequents in the conclusion and premise. The other kind of phase is the *synchronous* phase: here, rule applications of such inference rules often require choices. In particular, the introduction rule for the disjunction requires selecting either the left or right disjunct and the introduction rule for the existential quantifier requires selecting a term for instantiating the quantifier. The initial rule can terminate a synchronous phase and the cut rule can restart an asynchronous phase. Finally, there are three structural rules in *LKF*. The *store* rule recognizes that the first formula to the right of the $\Uparrow$ is either a negative atom or a positive formula: such a formula does not have an invertible inference rule and, hence, its treatment is delayed by storing it on the left. The *release* rule is used when the formula under focus (*i.e.*, the formula to the right of the $\Downarrow$) is no longer positive: at such a moment, the phase changes to the asynchronous phase. Finally, the *decide* rule is used at the end of the asynchronous phase to start a synchronous phase by selecting a previously stored positive formula as the new *focus*.

Impact of the polarity assignment. Let B be a first-order formula and let $\hat{B}$ result from B by placing either $+$ or $-$ on occurrences of t, f, $\wedge$, and $\vee$ (there are exponentially many such placements). It is proved in [15] that B is a classical theorem if and only if $\vdash \cdot \Uparrow \hat{B}$ has an *LKF* proof. Thus the different polarizations do not change *provability* but can radically change the structure of proofs. A simple induction on the structure of an *LKF* proof of $\vdash \cdot \Uparrow B$ (for some polarized formula B) reveals that every formula that occurs to the left of $\Uparrow$ or $\Downarrow$ in one of its sequents is either a negative literal or a positive formula. Also, it is immediate that the only occurrence of a contraction rule is within the decide rule: thus, only the positive formulas are contracted. Since there is flexibility in how formulas are polarized, the choice of polarization can, at times, lead to greatly reduced opportunities for contraction. When one is able to eliminate or constrain contractions, naive proof search can sometimes become a decision procedure.

3 Software architecture

Of the many qualities that we might want for a proof checker—universality, flexibility, efficiency, *etc.*—the one quality on which no compromise is possible is that of *soundness*. If we cannot prove or forcefully argue for the soundness of our checkers, then this project is without *raison d'être*.

3.1 Programming language support

An early framework for building sound proof checkers was the "Logic of Computable Functions" (LCF) system of Gordon, Milner, and Wadsworth [12]. In that framework, the ML programming language was created in order to support the task of building and checking proofs in LCF with a computing facility that provided strong typing and the abstractions associated to higher-order programming and abstract datatypes. Given the design of ML, it was possible to declare a type of theorems, say, `thm`, and to admit certain functions that are allowed to build elements of type `thm` (these encode axioms and inference rules). These latter functions could then be bundled into an abstract datatype and the programming language would enforce that the only items that eventually were shown to have type `thm` were those that ultimately were constructed from the axioms and inference rules encoded into the theorem abstract datatype. Of course, trusting that a checker written in this approach to LCF meant also trusting that (1) ML had the *type preservation property* and (2) the language implementation was, in fact, correct for the intended semantics (*i.e.*, that the addition function translated to the intended addition function, etc.).

This ML/LCF approach to proof checking is based on the most simple notion of proof (variously named after Hilbert or Frege) as a linear sequence of formulas arising from axioms and applications of inference rules.

The material in Section 2 illustrates that there can be a great deal more to the structure of proof than is available in such linear proof structures. We are fortunate that in order to take advantage of that rich structure, we do not need

$$\forall\Theta\forall\Gamma.\ \texttt{async}(\Theta, [t^-|\Gamma]).$$
$$\forall\Theta\forall\Gamma\forall A\forall B.\ \texttt{async}(\Theta, [(A \wedge^- B)|\Gamma])\ \texttt{:-}\ \texttt{async}(\Theta, [A|\Gamma]),\ \texttt{async}(\Theta, [B|\Gamma]).$$
$$\forall\Theta\forall\Gamma\forall A\forall B.\ \texttt{sync}(\Theta, A \vee^+ B)\ \texttt{:-}\ \texttt{sync}(\Theta, A);\ \texttt{sync}(\Theta, B).$$
$$\forall\Theta\forall\Gamma\forall P.\ \texttt{async}(\Theta, [])\ \texttt{:-}\ \texttt{memb}(P, \Theta),\ \texttt{pos}(P),\ \texttt{sync}(\Theta, P).$$
$$\forall\Theta\forall B\forall C.\ \texttt{async}(\Theta, [])\ \texttt{:-}\ \texttt{negate}(B, C),\ \texttt{async}(\Theta, B),\ \texttt{async}(\Theta, C).$$

Fig. 2. Five logic programming clauses specifying *LKF* inference rules

to invent a meta-language (in the sense that ML was invented to support LCF): an appropriate meta-language already exists in the λProlog programming language [18]. In contrast to the functional programming language ML, λProlog is a logic programming language. Like ML, λProlog is also strongly typed and has both higher-order programming and abstract datatypes. λProlog has a number of features that should make it a superior proof checker when compared with ML. In particular, λProlog's operational semantics is based on search and backtracking: this is in contrast to the notion of exception handling that is part of the non-functional side of ML. Furthermore, λProlog comes with much more of logic built into the language: in particular, it contains a logically sound notion of unification and substitution for expressions involving bindings (these latter features of λProlog are not generally provided by Prolog).

Although we shall not assume that the reader is familiar with λProlog, familiarity with the general notions of logic programming is particularly relevant to proof checking. Notice that it is nearly immediate to write a logic program that captures the *LKF* proof system in Figure 1. First select two binary predicates, say $\texttt{async}(\cdot,\cdot)$ and $\texttt{sync}(\cdot,\cdot)$, denoting the $\Uparrow$ and $\Downarrow$ judgments. Second write one Horn clause for each inference rule: here the conclusion and the premises of a rule correspond to the head and the body of such a clause. (The declarative treatment of the inference rules involving the quantifiers is provided directly by λProlog.) Of the fourteen Horn clauses that correspond to the fourteen inference rules in Figure 1, five are illustrated in Figure 2: these clauses correspond to the introduction rules for t^-, $\wedge^-$, and $\vee^+$ as well as the decide and cut rules. Some additional predicates have been introduced to specify membership in a multiset, the negation of a formula, and determining if a given formula is positive or not.

The full program can easily be seen to be sound in the sense that the sequent $\vdash \cdot \Uparrow B$ has an *LKF* proof if the atom $\texttt{async}([], B)$ has a proof using this logic program. Using standard depth-first search strategies would result, however, in surprisingly few proofs of the atom $\texttt{async}([], B)$: the clauses specifying the cut rule and the decide rule would immediately result in looping computations. We present this logic program not to suggest that it is appropriate for proving theorems but to show how to modify it to make it into a flexible proof checker.

$$\frac{t_e(\Xi)}{\Xi \vdash \Theta \Downarrow t^+} \qquad \frac{\Xi_1 \vdash \Theta \Downarrow B_1 \quad \Xi_2 \vdash \Theta \Downarrow B_2 \quad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Downarrow B_1 \wedge^+ B_2}$$

$$\frac{\Xi' \vdash \Theta \Downarrow B_i \quad i \in \{1,2\} \quad \vee_e(\Xi, \Xi', i)}{\Xi \vdash \Theta \Downarrow B_1 \vee^+ B_2} \qquad \frac{\Xi' \vdash \Theta \Downarrow [t/x]B \quad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \Theta \Downarrow \exists x.B}$$

$$\frac{\Xi_1 \vdash \Theta \Uparrow B \quad \Xi_2 \vdash \Theta \Uparrow \neg B \quad cut_e(\Xi, \Theta, \Xi_1, \Xi_2, B)}{\Xi \vdash \Theta \Uparrow \cdot}\ cut$$

$$\frac{\Xi' \vdash \Theta \Uparrow N \quad release_e(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\ release \qquad \frac{init_e(\Xi, \Theta, l) \quad \langle l, \neg P_a \rangle \in \Theta}{\Xi \vdash \Theta \Downarrow P_a}\ init$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \quad decide_e(\Xi, \Theta, \Xi', l) \quad \langle l, P \rangle \in \Theta \quad \text{positive}(P)}{\Xi \vdash \Theta \Uparrow \cdot}\ decide$$

$$\frac{\Xi' \vdash \Theta \Uparrow \Gamma \quad f_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow f^-, \Gamma} \qquad \frac{\Xi_1 \vdash \Theta \Uparrow A, \Gamma \quad \Xi_2 \vdash \Theta \Uparrow B, \Gamma \quad \wedge_c(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash \Theta \Uparrow A \wedge^- B, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \vee_c(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^- B, \Gamma} \qquad \frac{\Xi' \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \forall_c(\Xi, \Xi') \ \ y \text{ not free in } \Xi, \Theta, \Gamma, B}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma}$$

$$\frac{}{\Xi \vdash \Theta \Uparrow t^-, \Gamma} \qquad \frac{\Xi' \vdash \Theta, \langle l, C \rangle \Uparrow \Gamma \quad store_c(\Xi, C, \Xi', l)}{\Xi \vdash \Theta \Uparrow C, \Gamma}\ store$$

Fig. 3. The augmented *LKF* proof system LKF^a.

3.2 Clerks and experts

Consider being in possession of a proof certificate of a theorem and being asked to build an *LKF* proof of that theorem. The construction of the asynchronous phase is independent of any proof evidence you have (hence the name "asynchronous" for this phase). At the end of the asynchronous phase, the construction of the *LKF* proof can proceed with either the cut rule or the decide rule: in both cases, genuine information (a cut formula or a focus formula) must be communicated to the checker. Furthermore, the synchronous phase needs to determine which disjunct to discard in the $\vee^+$ rule and which term to use in the $\exists$ rule. To capture this sense of information flowing between a checker and a certificate, we present in Figure 3 an augmented version of *LKF*, called LKF^a. The augmentations to the *LKF* inference rules is done in three simple steps: (*i*) a proof certificate term, denoted by the syntactic variable Ξ is added to every sequent; (*ii*) every inference rule of *LKF* is given an additional premise using either an *expert predicate* or a *clerk predicate*; and (*iii*) the multiset of formulas to the left of the arrows $\Uparrow$ and $\Downarrow$ are extended to be a multiset of pairs of an *index* and a formula. Thus, the *LKF* proof system can be recovered from LKF^a by removing all occurrences of the syntactic variable Ξ and by removing all premises with a subscripted e or c as well as replacing all occurrences of tuples such as $\langle l, B \rangle$ with just B.

The expert predicates are used to mediate between the needs for information of the cut rule and the synchronous phase and the information that is present in a proof certificate. All of them examine the certificate Ξ and returns the information needed to continue with as many certificates as there are premises in the rules. For example, the disjunction expert returns either 1 or 2 depending

on which disjunct this introduction rule should select. The intension of the existential quantifier expert is that it returns a term t that is to be used in this introduction rule. Notice that the conjunction expert does nothing more than determine the proof certificates to be used in its two premises. The expert for the t^+ determines whether or not it should allow the proof checking process to end with this inference rule. The cut expert examines both the proof certificate and the context Θ and extracts the necessary cut formula for that inference rule. Notice that if this predicate is defined to always fail (*i.e.*, it is the empty relation), then checking this certificate will involve only cut-free *LKF* proofs. Finally, the decide expert gives the positive formula with which to start the new asynchronous phase.

The introduction rules of the asynchronous phase are given an additional premise that involves a clerk predicate: these new premises do not extract any information from the certificate but rather they take care of bookkeeping calculations involving the progress of the asynchronous phase. For example, the $\wedge_c(\Xi, \Xi_1, \Xi_2)$ judgment can be used to record in Ξ_1 the fact that proof checking is on the left branch of this conjunction as opposed to the right branch.

One of the strengths of our approach to proof certificates is that experts can be non-deterministic since this allows a trade-off between the size of a certificate and proof-reconstruction time. For example, let Ξ be a particular certificate and consider using it to introduce an existential quantifier. This introduction rule queries the expert $\exists_e(\Xi, \Xi', t)$. If the Ξ certificate explicitly contains the term t, the expert can extract it for use in this inference rules. If the certificate does not contain this term then the judgment $\exists_e(\Xi, \Xi', t)$ could succeed for every term t (and for some Ξ'). In this case, the expert provides no information as to which substitution term to use and, therefore, the certificate can be smaller since it does not need to contain the (potentially large) term t. On the other hand, the checker will need to reconstruct an appropriate such term during the checking process (using, for example, the underlying logic programming mechanism of unification). When experts are queried during the synchronous phase, their answers may be specific, partial, or completely unconstrained.

The three remaining rules (store, init, decide) of LKF^a reveal the structure of the collection of formulas we have been designating with the syntactic variable Θ. In our presentation of the *LKF* proof system, this structure has been taken to be a multiset of formulas. In our augmented proof system, we shall take this sequent context to be a multiset of pairs $\langle I, C\rangle$ where C is a formula and I is an index. When we need to refer to a specific occurrence of a formula in Θ (in, say, the decide rule), an index is used for this purpose. It is the clerk predicate associated to the store inference rule that is responsible for computing the index of the formula when it is moved from the right to the left of the $\Uparrow$. When the expert predicate in the decide rule describes the formula on which to focus, it does so by returning that formula's index. Finally, the initial expert determines which stored negative literal should be the complement of the focused literal. In the augmented form of both the decide and initial rules, additional premises have been added to check that the indexes returned by the expert predicates

are, indeed, indexes for the correct kind of formula: in this way, a badly defined expert cannot lead to the construction of an illegal *LKF* proof.

The structure of the indexing scheme is left open for the certificate definition to describe. As we shall illustrate later, indexes can be based on, for example, de Bruijn numbers, path addresses within a formula, or formulas themselves. It is possible for a formula to occur twice in the context Θ with two different indexes. We shall generally assume, however, that the indexes *functionally determine* formulas: if $\langle l, C_1\rangle \in \Theta$ and $\langle l, C_2\rangle \in \Theta$ then C_1 and C_2 are equal.

Assume that we have a logic programming system that provides a sound implementation of Horn clauses (for example, unification contains the occurs-check). A proof of $\Xi \vdash \cdot \Uparrow B$ within a logic programming implementation of LKF^a (along with the programs defining the experts and clerks) immediately yields an *LKF* proof of $\vdash \cdot \Uparrow B$. This follows easily since the logic programming proof of this goal can be mapped to an *LKF* proof directly: the only subtlety being that the mapping from indexes to formulas must be functional so that the indexes returned by the decide and initial rules are given a unique interpretation in the *LKF* proof. Notice that no such *LKF* proof is actually constructed: rather, it is performed. Notice also that this soundness guarantee holds with no restrictions placed on the implementation of the clerk and expert predicates.

3.3 Defining a proof certificate definition

In order to define a proof certificate for a particular format, we first need to translate theorems into LKU formulas. This operation stays outside the kernel and its correctness has to be proved. Furthermore we need to define the specific items that are used to augment *LKF*. In particular, the constructors for proof certificate terms and for indexes must be provided: this is done in λProlog by declaring constructors of the types `cert` and `index`. In addition, the definition must supply the logic program defining the clerk predicates and the expert predicates. Writing no specification for a given predicate defines that predicate to hold for no list of arguments. Figures 4, 5, and 6 are examples of such proof certificate definitions.

4 Some certificate definitions for classical logic

We now present some proof certificate definitions for classical logic: the first two deal with propositional logic while the third additionally treats first-order quantification. The first step is to define a translation function from classical formulas to LKF formulas. In this case, this boils down to choosing a polarization of the logical connectives and atomic formulas. Our first two examples of proof certificates are based on assigning negative polarizations to all atoms and to all connectives: *i.e.*, we only use $\wedge^-$, $\vee^-$, t^-, and f^-. A useful measurement of an *LKF* proof is its *decide depth*, *i.e.*, the maximum number of instances of the decide rule along any path from the proof's root to one of its leaves.

$$\begin{array}{rl}
\multicolumn{2}{c}{\texttt{cnf : cert}} \\
\forall C.\ \mathit{store}_c(\texttt{cnf}, C, \texttt{cnf}, \texttt{idx}(C)). \\
\forall \Theta \forall l.\ \mathit{init}_e(\texttt{cnf}, \Theta, l). \\
\forall \Theta \forall l.\ \mathit{decide}_e(\texttt{cnf}, \Theta, \texttt{cnf}, l). \\
\mathit{release}_e(\texttt{cnf}, \texttt{cnf}).
\end{array}
\qquad
\begin{array}{rl}
\multicolumn{2}{c}{\texttt{idx : form -> index}} \\
\wedge_c(\texttt{cnf}, \texttt{cnf}, \texttt{cnf}). \\
\vee_c(\texttt{cnf}, \texttt{cnf}). \\
f_c(\texttt{cnf}, \texttt{cnf}).
\end{array}$$

Fig. 4. A checker based on a simple decision procedure

4.1 A decision procedure

There is a simple decision procedure for checking whether or not a classical propositional formula is a tautology and we can design a proof certificate definition that implements such a decision procedure. This example illustrates an extreme trade-off between certificate size (here, constant-size) and proof reconstruction time (exponential time). In particular, notice that there is an *LKF* proof of a propositional formula if and only if that proof has decide depth 1 (possibly 0 if the formula contains no literals). The structure of an *LKF* proof of a tautology first builds the asynchronous phase, which ends with several premises all of the form $\vdash \mathcal{L} \Uparrow \cdot$ for some multiset of literals $\mathcal{L}$. Such a sequent is provable if and only if $\mathcal{L}$ has complementary literals: in that case, the *LKF* proof is composed of a decide rule (selecting a positive literal) and initial (matching that atom with a negative literal).

This decision procedure can be specified as the proof certificate definition in Figure 4. The single constant `cnf` is used for the certificate and formulas are used to denote indexes (thereby trivializing the notion of indexes) so we need a constructor to coerce formulas into indexes. Figure 4 also contains the specifications of the clerk and expert predicates. Notice that the initial expert does not behave expertly: it relates the `cnf` certificate to all indexes l and all contexts Θ. Our definition of this predicate here can be unconstrained since the index that it returns is not trusted: that is, the initial rule in LKF^a will check that l is the index of the complement of the focus formula. In the usual logic programming sense, the *check* in the premise is all that is necessary to *select* the correct index. A similar statement holds for the decide expert predicate definition.

4.2 Matings

Let B be a classical propositional formula in negation normal form. Andrews defined a *mating* $\mathcal{M}$ for B as a set of complementary pairs of literal occurrences in B [2]. A mating denotes a proof if every *vertical path* in B (read: clause in the conjunctive normal form of B) contains a pair of literal occurrences given by set $\mathcal{M}$. A certificate definition for proof matings is given in Figure 5. Indexes are, in fact, paths in a formula since they form a series of instructions to move left or right through the binary connectives or to stop (presumably at a literal). There are two constructors for the `cert` type: `aphase` is applied to a list of indexes and `sphase` is applied to a single index. These two constructors are used to mimic

$$
\begin{array}{rl}
\texttt{root} : \texttt{index} & \texttt{left},\ \texttt{right} : \texttt{index} \to \texttt{index} \\
\texttt{aphase} : \texttt{list index} \to \texttt{cert} & \texttt{sphase} : \texttt{index} \to \texttt{cert} \\
\forall I \forall Is.\ \vee_c(\texttt{aphase}([I|Is]), \texttt{aphase}([\texttt{left}(I), \texttt{right}(I)|Is])). \\
\forall I \forall Is.\ \wedge_c(\texttt{aphase}([I|Is]), \texttt{aphase}([\texttt{left}(I)|Is]), \texttt{aphase}([\texttt{right}(I)|Is])). \\
\forall I \forall Is.\ f_c(\texttt{aphase}([I|Is]), \texttt{aphase}(Is)). \\
\forall C \forall I \forall Is.\ \mathit{store}_c(\texttt{aphase}([I|Is]), C, \texttt{aphase}(Is), I). \\
\forall I.\ \mathit{release}_e(\texttt{sphase}(I), \texttt{aphase}([I])). \\
\forall \Theta \forall l.\ \mathit{decide}_e(\texttt{aphase}([]), \Theta, \texttt{sphase}(l), l) \\
\forall \Theta \forall k \forall l.\ \mathit{init}_e(\texttt{sphase}(k), \Theta, l)\ \texttt{:-}\ \ \langle k, l \rangle \in \mathcal{M}.
\end{array}
$$

Fig. 5. Mating certificate definition

the two kinds of sequents in *LKU*: `aphase` and `sphase` denote the formulas in the right-hand side of asynchronous and synchronous sequents by their paths from the root. The initial expert will only select index l if it is $\mathcal{M}$-mated to the focused formula (with path address k). Here, we have assumed that $\mathcal{M}$ contains ordered pairs of occurrences in which the first occurrence names a positive literal and the second occurrence names a negative literal. Thus, in order to determine if $\mathcal{M}$ is a proof mating for the formula B, set $\hat{B}$ to be the polarization of B using only negative connectives and check that the certificate $\texttt{aphase}([root])$ can lead the clerks and experts in Figure 5 to a successful execution with $\hat{B}$.

4.3 Resolution refutations

A (resolution) clause is a closed formula that is the universal closure of a disjunction of literals (the empty disjunction is false). When we polarize, we use the negative versions of these connectives and we assign negative polarity to atomic formulas. We assume that a certificate for resolution contains the following items: a list of all clauses $C_1, \ldots, C_p$ ($p \geq 0$); the number $n \geq 0$ which selects the last clause that is part of the original problem (*i.e.*, this certificate is claiming that $\neg C_1 \vee \cdots \vee \neg C_n$ is provable and that $C_{n+1} ... C_p$ are intermediate clauses used to derive the empty one); and a list of triples $\langle i, j, k \rangle$ where each such triple claims that C_k is a binary resolution (with factoring) of C_i and C_j. If the implementer of a resolution prover wished to output refutations, this kind of document should be easy to accommodate.

Checking this structure is done in two steps. First, we check that a particular binary resolution is sound and then we check that the list of resolvents leads to an empty clause. It is a simple matter to prove the following: if clauses C_1 and C_2 yield resolvent C_0 as a binary resolvent (allowing also factoring), then the focused sequent $\vdash\ \neg C_1, \neg C_2 \Uparrow C_0$ has a proof of decide depth 3 or less. We can also restrict such a proof so that along any path from the root sequent to its leaves, the same clause is not decided on more than once. The first part of Figure 6 contains the ingredients of a checker for the claim $\vdash\ \neg C_1, \neg C_2 \Uparrow C_0$. This checking uses two constructors for indexes. The first is used to reference clauses (*i.e.*, the expression $\texttt{idx}(i)$ denotes $\neg C_i$) and the second constructor is

$$\texttt{idx} : \texttt{int} \to \texttt{index} \qquad \texttt{lit} : \texttt{form} \to \texttt{index}$$
$$\texttt{dl} : \texttt{list int} \to \texttt{cert} \qquad \texttt{ddone} : \texttt{cert}$$

$$\forall L.\ \vee_c(\texttt{dl}(L), \texttt{dl}(L)). \qquad \forall L.\ t_e(\texttt{dl}(L)).$$
$$\forall L.\ f_c(\texttt{dl}(L), \texttt{dl}(L)). \qquad \forall L.\ \forall_c(\texttt{dl}(L), \texttt{dl}(L)).$$
$$\forall C \forall L.\ store_c(\texttt{dl}(L), C, \texttt{dl}(L), \texttt{lit}(C)). \qquad \forall L.\ \exists_e(\texttt{dl}(L), \texttt{dl}(L), T).$$
$$\forall L \forall P \forall \Theta.\ decide_e(\texttt{dl}(L), \Theta, \texttt{ddone}, \texttt{lit}(P)). \qquad \forall L.\ \wedge_e(\texttt{dl}(L), \texttt{dl}(L), \texttt{dl}(L)).$$
$$\forall I \forall \Theta.\ decide_e(\texttt{dl}([I]), \Theta, \texttt{dl}([]), \texttt{idx}(I)). \qquad \forall l \forall \Theta.\ init_e(\texttt{ddone}, \Theta, l).$$
$$\forall I \forall J \forall \Theta.\ decide_e(\texttt{dl}([I, J]), \Theta, \texttt{dl}([J]), \texttt{idx}(I)). \qquad \forall l \forall L \forall \Theta.\ init_e(\texttt{dl}(L), \Theta, l).$$
$$\forall I \forall J \forall \Theta.\ decide_e(\texttt{dl}([J, I]), \Theta, \texttt{dl}([J]), \texttt{idx}(I)). \qquad \forall L.\ release_e(\texttt{dl}(L), \texttt{dl}(L)).$$

$$\texttt{rdone} : \texttt{cert} \qquad \texttt{rlist} : \texttt{list (int * int * int)} \to \texttt{cert}$$
$$\texttt{rlisti} : \texttt{int} \to \texttt{list (int * int * int)} \to \texttt{cert}$$
$$\forall R.\ f_c(\texttt{rlist}(R), \texttt{rlist}(R)).$$
$$\forall C \forall l \forall R.\ store_c(\texttt{rlisti}(l, R), C, \texttt{rlist}(R), \texttt{idx}(l)).$$
$$t_e(\texttt{rdone}).$$
$$\forall I \forall \Theta.\ decide_e(\texttt{rlist}([]), \Theta, \texttt{rdone}, \texttt{idx}(I)) \ \texttt{:-}\ \langle \texttt{idx}(I), t \rangle \in \Theta.$$
$$\forall I, J, K, R, C, N, \Theta.\ cut_e(\texttt{rlist}([\langle I, J, K \rangle | R]), \Theta, \texttt{dl}([I, J]), \texttt{rlisti}(K, R), N) \ \texttt{:-}$$
$$\langle \texttt{idx}(K), C \rangle \in \Theta,\ \texttt{negate}(C, N).$$

Fig. 6. Resolution certificate definition in two parts

used to index literals that need to be stored: here the literal is used to provide its own index. The first two `cert` constructors in that figure are used to control the sequencing of decide rules involving two (negated) clauses. The first of these constructors provides the sequent of clause indexes (at most 2) used to build a proof and the second constructor is used to signal that the proof should finish with the selection of stored literals and not with additional clauses.

The clerks for this part of the checking process do essentially no computation and just move certificates around unchanged: the exception is the store clerk that provides the trivial index $\texttt{lit}(C)$ for the literal C. The only expert that provides information to guide proof reconstruction is the decide expert which transforms the choice of clauses to consider from two to one to none. Given these clerks and experts, it is now the case that if C_i and C_j resolve to yield C_k then $\texttt{dl}([i, j]) \vdash \neg C_1, \ldots, \neg C_m \Uparrow C_k$ is provable. With only small changes, the binary resolution checker can be extended to hyperresolution: in this case, the experts will need to attempt to find a proof of decide depth $n + 1$ when attempting to resolve together $n \geq 2$ clauses.

To describe a checker for a complete certificate, we use three additional constructors for certificates as well as the additional clauses in the second part of Figure 6. Notice that the decide expert only proposes a focus at the end of the checking process when the list of triples (resolvents) is empty: this expert only succeeds if one of the clauses is t (the negation of the empty clause). It is the cut expert that is responsible for looping over all the triples encoding resolvents. Notice that the cut-formula is the clause C_k and that the left premise invokes

the resolvent checking mechanism described above. The right premise of the cut carries with it an index (in this case, k) so that the next step in the proof checking knows which index to use to correctly store that formula. The *LKF* proof that is implicitly built during the checking of a resolution contains one cut rule for every resolvent triple in the certificate.

4.4 Capturing general computation within proofs

The line between computation and deduction is certainly movable and one that a flexibly designed proof certificate definition should allow to be moved. As we saw in Section 4.1, we can use naive proof reconstruction to compute, for example, the conjunctive normal form of a propositional formula. We can go further, however, and allow for arbitrary Horn clause programs to be computed on first-order terms during proof reconstruction. For example, if one needs to check a proof rule that involves a premise that requires one number to divide another number, it is an easy matter to write a (pure) Prolog program that computes this binary relationship on numbers. Such Horn clauses can be added to the sequent context and a proof certificate could easily guide the construction of a proof of that premise from such clauses.

5 Adequacy of encoding

Our use of the augmented *LKF* proof system as our kernel guarantees soundness no matter how the clerk and expert predicates are defined. On the other hand, one might want to know if the checker is really checking the proof intended in the certificate. A checker for a mating could, in fact, ignore the mating and run the decision procedure from Section 4.1 instead. The kernel itself cannot guarantee the *adequacy* of the checking: knowledge of the certificate definition is necessary to ensure that. As our examples show, however, the semantics of the clerk and expert predicates is clearly given by the LKF^a proof system and certificate definitions are compact: thus, verifying certificates should be straightforward.

Some aspects of a proof certificate are not possible to check using our kernel. Consider defining a *minimal proof mating* to be a proof mating for which no mated pairs can be removed and still remain a proof mating. We see no way to capture this minimality condition: that is, we see no way to write a certificate definition that successfully approves a mating if and only if it is a minimal proof mating. A similar observation can be made with resolution: if $\vdash \neg C_1, \neg C_2 \Uparrow C_0$ has a proof (even a proof of decide depth 3) it is not necessarily the case that C_0 is the resolvent of C_1 and C_2. For example, the resolution of $\forall x[p(x) \vee r(f(x))]$ and $\forall x[\neg p(f(x)) \vee q(x)]$ is $\forall x[r(f(f(x))) \vee q(x)]$. At the same time, it is possible to prove the sequent

$$\vdash \exists x[\neg p(x) \wedge \neg r(f(x))], \exists x[p(f(x)) \wedge \neg q(x)] \Uparrow \forall x[r(f(f(f(x)))) \vee q(f(x)) \vee s(f(x))].$$

This formula is similar to a resolvent except it uses a unifier that is not most general and it has an additional literal. Thus, when this check succeeds, what is checked is its soundness and not its technical status of being a resolvent.

6 The more general kernel

As we have mentioned, a more general kernel for proof checking is based not on *LKF* but the *LKU* proof system [16]. Instead of the two polarities in *LKF*, there are four polarities in *LKU*: the polarities -1 and $+1$ denote positive and negative polarities of linear logic while the polarities -2 and $+2$ denote the positive and negative polarities of classical logic. Intuitionistic logic use formulas that have subformulas of all four polarities. In order to restrict the *LKU* proof system to emulate, say, *LKF* or *LJF*, one simply needs to describe certain restrictions to the structural rules (store, decide, release, and init) of *LKU*. The logic definition documents (see Section 1) declare these restrictions.

The *LKU* proof system makes it possible to use the vocabulary for structuring checkers in *LKF* (clerks, experts, store, decide, release) to also design checkers in the intuitionistic focused framework *LJF*. The main subtleties with using *LKU* is that we must deal with a linear logic context: since such contexts must be *split* into two contexts occasionally, some of the expert predicates need to describe which splitting is required. We have defined certificate definitions for simple and dependent typed λ-calculus: that is, the *LKU* kernel can check natural deduction proofs in propositional and first-order intuitionistic logic (de Bruijn numerals make a natural index for store/decide).

7 Related and future work

The first mechanical proof checker was de Bruijn's Automath [8] which was able to check significant mathematical proofs. As we have mentioned in Section 3, another early proof checker was the ML implementation of LCF's tactics and tacticals [12] (for a λProlog implementation of these, see [18]). As the number and scope of mechanical theorem proving systems has grown, so too has the need to have one prover rely on other provers. For example, the OpenTheory project [14] aims at having various HOL theorem provers share proofs. Still other projects attempt to connect SAT/SMT systems with more general theorem provers, *e.g.*, [3, 6, 9]. In order for prover A to not blindly trust proofs from prover B, prover B may be required to generate a certificate that demonstrates that it has formally found a proof. Prover A will then need to check the correctness of that certificate. In this way, prover A only needs to check individual certificates and not rely on trusting the whole of prover B. Of course, every pair of communicating provers could involve certificates of different formats and different certificate checker. Our goal here is to base such certificates on *foundational* and *proof-theoretic* principles and to describe programmable checkers that are guaranteed to be sound. Also, since that checker is based on well understood and well explored declarative concepts (*e.g.*, sequent calculus, unification, and backtracking search), that checker can be given many different implementations.

The Dedukti proof checker [5] implements $\lambda\Pi$ *modulo*, a dependently typed λ-calculus with functional rewriting. Given a result of Cousineau & Dowek [7] that any functional Pure Type System can be encoded into $\lambda\Pi$ modulo, Dedukti

can check proofs in such type systems. As we have described above, the proof certificate setting described here allows one to capture both dependently typed λ-terms and computations (not just functional computations). As a result, we should be able to design, following [7], proof certificates for pure type systems. The dependently typed λ-calculus LF has recently been extended to LFSC [20] and to $\mathrm{LF}_{\mathcal{P}}$ [13] so that various kinds of computations can be treated by the type checker instead of being explicitly detailed within the typed λ-term itself. Such proof objects should similarly be captured in our setting.

Getting provers to trust each other's proofs using the techniques described in this paper will require the development and acceptance of an infrastructure and associated tools, something that can clearly take time. One area where proof certificates can make an early impact is in theorem proving competitions. In such competitions, theorem provers should not be trusted but rather the proof certificates that they emit should be checked. In that case, our framework for foundational proof certificates can provide a clear semantics for what constitutes a proof certificate.

Besides the proof certificates definitions that we have described above, we have designed other examples (including proof nets for multiplicative linear logic and Frege proofs) and plan to develop more. This work on foundational proof certificates is part of a more ambitious project to design proof certificates that also allow for induction and coinduction: such certificates should allow model checkers and inductive theorem provers to communicate with each other. We also hope to eventually allow counterexamples to be checked and to interact with (partial) proofs [17].

We have only considered the problem of communicating and checking formal proofs between machines. Of course, proofs are important to humans as well. Given the fact that proof certificates can be elaborated into a *LKU* sequent proof, it might well be possible to use proof-theoretic results to construct tools that allow humans to browse and interact with formal proofs in order to learn from them. We leave such considerations for future work.

8 Conclusion

In a world where proof certificates can be designed flexibly and given precise semantics and where proof checkers can be given a high degree of trust, the sharing of proofs should become "feature zero" for all new theorem provers. That is, implementers looking to get their provers accepted broadly will need to first consider how to communicate their proof evidence as a checkable certificate. In such a world, proofs can be liberated from the technologies that produced them (*e.g.*, Coq, Isabelle, and Mizar) and can be seen as the universal and eternal objects logicians and proof theorists have long been working to place at the foundations of mathematics and computer science.

Acknowledgments: We thank Jean Pichon, Thanos Tsouanas, and the reviewers for their comments on an earlier draft of this paper. This work was funded by the ERC Advanced Grant ProofCert.

References

1. J-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. P. B. Andrews. Theorem-proving via general matings. *J. ACM*, 28:193–214, 1981.
3. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jouannaud and Shao, eds, *Certified Programs and Proofs*, LNCS 7086, 135–150, 2011.
4. D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), April 2012.
5. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. *Proof Exchange for Theorem Proving*, 28–43, 2012.
6. S. Böhme and T. Weber. Designing proof formats: A user's perspective. *Proof eXchange for Theorem Proving*, 27–32, August 2011.
7. D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. *TLCA 2007*, LNCS 4583, 102–117, Springer, 2007.
8. N. G. de Bruijn. Reflections on Automath. In R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, 201–228. North-Holland, 1994.
9. P. Fontaine, J-Y. Marion, S. Merz, L. P. Nieto, and A Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. *TACAS 2006*, LNCS 3920, 167–181. Springer, 2006.
10. G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, 68–131. North-Holland, 1969.
11. J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
12. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, LNCS 79. Springer, 1979.
13. F. Honsell, M. Lenisa, L. Liquori, P. Maksimovic, and I. Scagnetto. $\mathrm{LF}_{\mathcal{P}}$: a logical framework with external predicates. In *LFMTP'12: Proceedings of the seventh international workshop on Logical frameworks and meta-languages, theory and practice*, pages 13–22. ACM New York, 2012.
14. J. Hurd. The OpenTheory standard theory library. In *Third International Symposium on NASA Formal Methods*, LNCS 6617, 177–191, 2011.
15. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
16. C. Liang and D. Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011.
17. D. Miller. A proposal for broad spectrum proof certificates. *CPP: First International Conference on Certified Programs and Proofs*, LNCS 7086, 54–69, 2011.
18. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
19. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
20. A. Stump. Proof checking technology for satisfiability modulo theories. Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP), 2008.